2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

A Self-Adaptable Approach for Easing the Development of Grid-Oriented Services

André Lage Freitas and Jean-Louis Pazat
INRIA / IRISA – MYRIADS Team
Campus de Beaulieu
35 042 Rennes, France
Emails: {Andre.Lage,Jean-Louis.Pazat}@irisa.fr

***Abstract*—The Service-Oriented Architecture (SOA) leverages the service abstraction to enable the development of modular, loose-coupled and distributed applications. In order to use such an architecture, service-based applications directly rely on services or compose them for conceiving new functionalities. In spite of these capabilities, they do not support the development of services which need high-performance computing. Grid computing offers an infrastructure for high-performance computing which is based on the sharing of distributed, low-cost and heterogeneous resources in large-scale. Thus, grids can be used to satisfy these high-performance service requirements. This work aims at easing the development of grid-oriented services. The iPOJO service-component model is used to propose an architecture that automatically manages job submission for services. This architecture is based on Dynaco (Adaptation for Components) and the XtreemOS grid operating system.**

I. Introduction

The *Service-Oriented Computing* (SOC) [17] uses the service abstraction to address the development of modular and loose-coupling distributed applications. These service-based applications directly use atomic services or compose them in order to obtain other functionalities. The relationship among services are defined and established through agreements which are formalized as contracts. In order to enable the development of such applications, the Service-Oriented Architecture (SOA) proposes a support which addresses service composition as well as basic and high-level service management features. However, the SOA was not conceived to take into account non-trivial computational capabilities for services that require high-performance computing.

Grid computing [7] leverages low-cost and heterogeneous resources in order to provide a distributed infrastructure for high-performance computing in large-scale. Grids precisely define resource access through the management of Virtual Organizations (VO) in a dynamic fashion. Thereby, grids offer an infrastructure suitable for supporting services that have high-performance computing needs. In order to use the grid, such services rely on the grid job abstraction by submitting and managing them through the grid interface. However, in spite of interesting efforts as the Simple Grid API (SAGA) [8] that addresses to ease the use of grids, grid usage still remains complex. This drawback becomes an inconvenience for the development of grid-oriented services.

This work proposes a self-adaptable support for easing the conception of grid-oriented services. The self-adaptive aspect is addressed by Dynaco (Dynamic Adaptation for Components) [3] in a design based on components. We rely on iPOJO [6] due to its dynamic and flexible capabilities for developing and maintaining services on OSGi [14] platforms. Finally, a self-adaptable iPOJO handler is proposed which addresses the management of jobs on the XtreemOS grid operating system.

This work is organized as follows. Section II presents the Service-Oriented Computing and iPOJO. Grid Computing is discussed in Section III. In Section IV, we introduce dynamic adaptation and the Dynaco adaptation model. Our design is proposed in Section IV and it is followed in Section V by an architecture that describes the enabling technologies for such a proposal. We discuss about related works in Section VII. Ultimately, we conclude our discussion in Section VIII and we present our future plans concerning our investigation in this field.

II. Service-Oriented Computing

A. Overview

The *Service-Oriented Computing* (SOC) [17] paradigm proposes a modular and loose-couple design for developing distributed applications. Such a design is based on the service abstraction which uses contracts to define how these service will interact with each other. The SOC is represented by the Service-Oriented Architecture (SOA). The SOA describes a three-layer architecture widely known as the SOA pyramid which enables the conception of service-based applications. In a brief, the lowest SOA layer addresses basic functionalities as service discovery and binding. The middle-layer deals with service composition in which services are combined in order to conceive composite services. Last, the upper layer handles high-level service management features that are mainly related to service life cycle.

There exists different understandings of the SOA which are usually defined by open technical specifications. Firstly, we can evidence Web Services (WS) [19] due to its wide adoption. Web Services main address the integration of distinct systems by using Web standards as UDDI, XML, HTTP and SOAP. Secondly, the Service-Component Architecture (SCA) [13] specification addresses a uniform and structured way of developing services by means of defining a component model. Such a model enables component instances to implement services

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.56

through specific bindings. Finally, different from the formers, OSGi [14] addresses the management of service dynamism by proposing a flexible SOA platform framework. It is further explained as follows.

B. OSGi

The OSGi Alliance was created by a group of industries in 1999 targeting to define open specifications for services that operate on local network devices. Currently, it still aims at providing open service specifications but addressing Internet services that include desktops, homes, cars and ubiquitous devices. The OSGi framework relies on the Java technology and targets to ease the development of services that operate on "small-memory devices that can be deployed on large scale" [14]. However, there are current efforts exposed in [16] address this issue by decentralizing the platform and providing interoperability with other service technologies.

The OSGi specification is based on a modular design that takes advantage of deployment units called *bundles* which implement OSGi services. OSGi bundles communicate to each other in a dynamic fashion by exporting and importing Java packages. A bundle can contain one or more services that either provide or require other services. In OSGi, service descriptors are defined by Java interfaces. The OSGi framework offers then a simple and efficient way of managing bundles and their services which facilitates the application management and maintenance. However, though the OSGi framework provides such advantages, it is still the service developer charge to deal with this dynamism.

C. iPOJO

There has been proposed by [4], [20] the use of the component-based design [18] to develop services due to their complementary characteristics. On one hand, services are loose-coupling, dynamic and business-oriented. On the other hand, components offer an efficient development model that separates concerns and promotes re-usability. These properties simplify the service development. Moreover, with regard to OSGi platform, component-oriented service models such as Declarative Services [15] and iPOJO [6] do not only facilitate the service development, but they also support the management of the OSGi dynamism which is often addressed by service developers.

Among approaches that offer a component model for OSGi, iPOJO stands out due to two main aspects. First, it provides structured component composition through composites whose instances can be visualized as an architectural view of the service. Second, iPOJO's dynamism management separates the functional and non-functional service codes which are respectively represented by POJOs[1] (content) and handlers (controllers). Thus, service developers may focus on the service main development while iPOJO handlers deal with service non-functional requirements. Such non-functional requirements may concern details about servicing and requiring services or triggering actions according to life cycle callbacks, for instance. Moreover, iPOJO provides extended handlers that offer non-trivial features as support for design patterns, event-based communication and time-based service requiring. Finally, further handlers can be implemented thanks to extensible features in iPOJO's design and implementation.

[1]Plain Old Java Object.

III. Grid Computing

A. Overview

Grid computing [7] addresses the use of distributed and low-cost resources for performing high-performance computing. Even though such resources may be heterogeneous and geographically dispersed, grids target to share them in a transparent fashion. In order to ensure such a transparency, grids rely on open standards that unify the way of how grid resources are used. Furthermore, concerning the resource access policies, grids leverage the concept of Virtual Organization (VO) to define and ensure user privileges in each resource. Thereby, distinct organizations or different sectors of a organization can configure the grid usage to suit their needs.

Grid applications take advantage of the *job* abstraction and may consist of one or more grid jobs. Jobs are commonly formed by a parallel program whose tasks may be related to each other or not. Moreover, job descriptions contain both the program binary and the resource requirements to execute it. These requirements may describe the number and type of resources and detailed information about them which may include operating system, libraries, input data, for instance. With respect to the job execution, grids propose distinct interfaces such as APIs, terminals and web portals through which jobs can be submitted and managed. Job management commands are dealt with by the grid *broker* which is in charge of allocating the resources according to the job requirements. Then the broker sends the job-resources mismatch to the grid *scheduler* in order to execute the job. Once the job finishes, the results are stored in the user's space in the *grid file system*.

In spite of the grid benefits earlier exposed, grid usage still remains complex. Moreover, each grid platform provides its own interface with customized commands. In order to ease the grid application development and to promote the interoperability among different grids, the Grid Simple API (SAGA) [8] proposes a standardized grid API. SAGA allows grid applications to be developed taking into account a unique and simple interface while it enables the execution of such an application in distinct grids. Moreover, SAGA achieves this goal by leveraging grid *adaptors* which can be developed under grid interfaces written in Java or C++ languages currently.

B. XtreemOS

The XtreemOS project proposes the idea of a grid operating system in contrast to conventional grid approaches based on grid middlewares. XtreemOS uses the Linux kernel to build specific modules which transparently enables Mandriva GNU/Linux PCs and mobile devices to become grid resources. Moreover, it also supports single-system image clusters which grid users can use as it would be a single and powerful

machine. With respect to resource management, XtreemOS takes advantage of such kernel implementation to decrease the system overhead when performing tasks that require detailed information of the system. Finally, XtreemOS interface (XOSAGA) is based on SAGA by providing higher-level abstractions for developing grid applications; besides XtreemOS also provides a POSIX-compliant terminal interface.

The design of the XtreemOS grid operating system is divided in *foundation* and *grid* layers. The former addresses the low-level kernel implementations that allow to use different hardwares as grid resources. Thus, the communication infrastructure is built under those kernel modules and allows both foundation and grid layers to interact with each other. The latter layer is responsible for providing grid functionalities. It consist of three entities that provide *Application Execution Management* (AEM), *Virtual Organization Management* (VOM) and the *XtreemOS File System* (XFS). Those functionalities are transparently offered as in conventional operating systems thanks to the foundation layer which offers such a transparency.

IV. Dynamic Adaptation

Systems that rely on dynamic environments should be able to adapt themselves in order to deal with unpredicted changes. Dynamic environments commonly present changes which are not able to predict at design time. These changes may lead systems to face undesirable situations which degrade and compromise the application working. Moreover, it is harder to both predict and deal with such changes in distributed scenarios. Therefore, the adaptation of applications at runtime becomes then a fundamental feature in order to ensure their proper behavior.

In [3], [10], [1], [12] there are further discussions about adaptation techniques. While [1], [12] address general issues and enabling technologies for software adaptation, [3], [10] propose an adaptation design that divides distinct adaptation concerns: *Monitor, Analyze, Plan* and *Execute*[2]. On one hand, in [10], the authors propose a general architecture for *Autonomic Computing*. On the other hand, Dynaco (Dynamic Adaptation for Components) [3] proposes an adaptation model validated by a framework for developing self-adaptable application on grids. Furthermore, Dynaco promotes the re-usability of adaptation generic mechanisms which can be customized according to the adaptation domain.

The Dynaco adaptation model proposes to separate the adaptation implementation from the application functional code. It leverages the component-based design by addressing the adaptation implementation as component controllers. Thus such a controller is based on an adaptation functionality decomposition which separates the four aforementioned adaptation aspects. These adaptation aspects are represented by the following entities whose relationships and further details are described as follows.

[2]Also known as MAPE cycle.

1) Monitoring: The *monitor* is in charge of gathering information that are related to adaptation interests. It relies on *pull* and *push* flows in order to either keep other entities aware about a change (i.e., event-based communication) or to let them ask about specif metric measures.

2) Decision: When a change occurs, the *decider* decides whether it is enough relevant for performing an adaptation or not. If so, it concerns about what should be be done by means of a *strategy* which contains the goal that should be achieved in order to change the application behavior to achieve a proper state. For instance, it could be *"load balancing among resources R1, R2 and R3"* because R1 presents performance degradation due to three concurrent processes *p1, p2* and *p3*.

3) Planning: Once the decider defines which strategy would be employed, it is sent to the *planner* that translates such a high-level goal to a set of instructions as a *plan*. According to the earlier example, the plan would be an instruction as *"migrate process p2 to R2 and p3 to R3"*.

4) Execution: Thereby, the plan is sent to the *executor* which performs the plan that will finally make the component adapt to a more suitable configuration. For achieving this, it must intercept the application flow execution and then execute the received instructions.

V. Towards a Self-Adaptable Support for Grid-Oriented Services

In order to use the grid, service developers must interact with the grid interface which is commonly employed through the grid API. Thus grid jobs can be submitted and managed by invoking respective methods. However, the task of managing jobs is complex, time consuming and error prone. We understand that service developers should be free from this task and finally focus on main aspects of the service development. Consequently, it claims a support which is able to transparently manage job submissions for services aiming at easing the development of grid-oriented services. Furthermore, such a support should consider QoS related to the job execution in order to proper satisfy service specific needs.

We can outline two technologies that enable the conception of automatic job management for grid-oriented services. The first one the component-based design which have been used to facilitate the development of services [6], [15], [20]. They allow to separate functional from non-functional services requirements which facilitates both service development and maintenance. Moreover, components do not only ease the development of services, but they also are a suitable technology to enable the job management for services. By using components, it is possible to separate the job management task from other interests related to the service development.

The second technology that enables to automatically manage jobs is self-adaptation. Even though grids offer a robust platform for executing complex applications, grids are dynamic distributed systems which face unforeseen changes. Some of these changes may affect the job execution in such a way that would require restarting or even resubmit the job. In order to deal with such a scenario, self-adaptive

techniques ought to be exploited when providing a support for job management. Finally, we combine components and self-adaptation as the foundation of a job self-management support for services. This support uses the Dynaco (Dynamic Adaptation for Components) adaptation model and it is further explained as follows.

A. Proposal Overview

Our goal is to ease the development of services that use grids by providing an integrated approach which automatically manages job submissions. Figure 1 summarizes and positions our proposal. Firstly, we tackle the composition layer of the SOA pyramid, more precisely services that need to execute grid jobs. Secondly, we rely on services which are based on components due to great advantages that such a design offers as earlier explained. Then we separately deal with the job management and the component functional concerns by addressing the former as a component controller and the latter in the component content. As follows, we use the Dynaco adaptation model to tackle the self-adaptive behavior of the job management support. Finally, the grid is used as the underlying infrastructure for executing the job. Furthermore, we assume that services have agreed on QoS values which is then used to guide the adaptation strategy. However, though agreement negotiation is taken into account, it is not in the scope of our proposal. In [9], it is further discussed how such a negotiation can be performed as well as the translation of SLA QoS to resource-level QoS related to the job execution.

VI. A Self-Adaptable iPOJO Handler for XtreemOS

A. Architecture

In this section it is presented an architecture that describes how the previous design proposed in V can be employed. First, as our goal is to ease the development of grid-oriented services, we propose to use technologies that facilitate such a development. They are explained as follows:

1) SOA Platform: We leverage the OSGi platform which proposes a dynamic and modular framework for developing and maintaining services.

2) Component-Oriented Service Model: iPOJO rightly supports the management of OSGi services. It provides a component model that dynamically manages service bindings while allowing to define details about both servicing and requiring services.

3) Adaptation Model: Dynaco proposes an adaptation model which separately addresses distinct adaptation aspects. It offers a clear design for developing self-adaptive techniques.

4) Grid Infrastructure: In order to have a deeper control of grid resources in a transparent way, we use the XtreemOS as grid infrastructure. It offers the XOSAGA higher-level interface and transparently deal with resource management.

Based on those technologies, we propose the design of an iPOJO handler that relies on the Dynaco adaptation model and performs job self-management on the XtreemOS grid

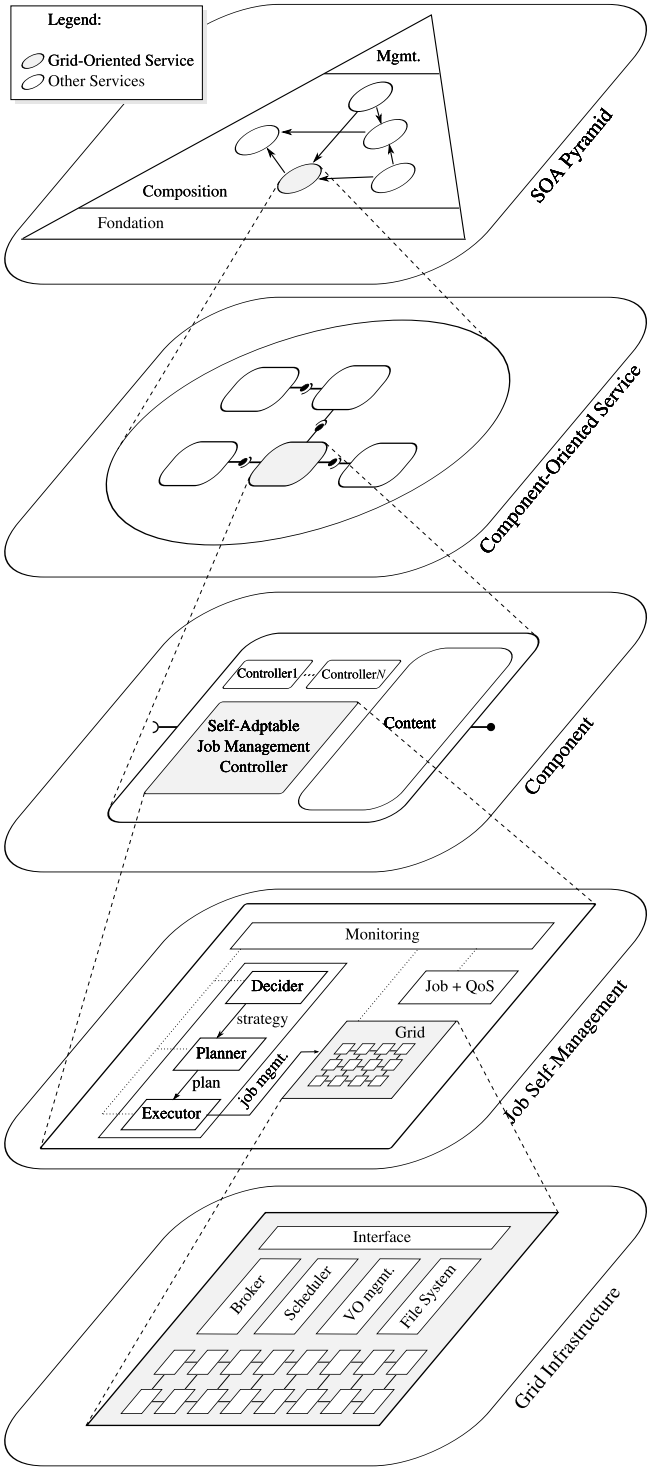

Fig. 1. A Self-Adaptable Support for Grid-Oriented Services.

operating system as depicted in Figure 2. The monitoring system uses the *iPOJO Event Admin* and has information about the jobs, their requirements and QoS through the component metadata file. Besides the monitoring system keeps the decider, planner and executor informed about these facts as well as about the grid environment. Finally, jobs are submitted and managed on XtreemOS through XOSAGA commands sent by the executor.

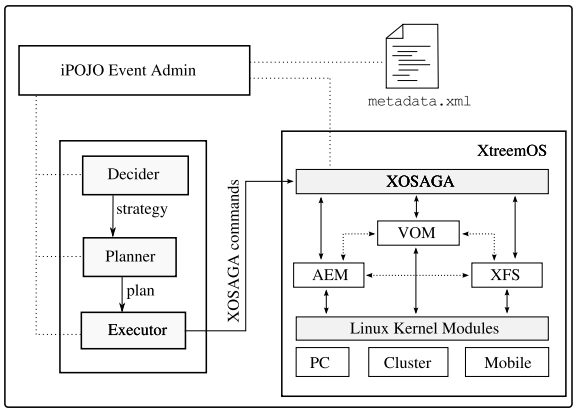

Fig. 2. The Self-Adaptable iPOJO Handle for Managing XtreemOS Jobs.

B. An iPOJO Handler for XtreemOS

We take advantage of the extensible iPOJO component model that allows to implement further iPOJO handlers. iPOJO handlers are implemented by extending the `PrimitiveHandler` abstract class as illustrated by Figure 3. The `PrimitiveHandler` class inherits the `FiledInterceptor` and the `MethodInterceptor` classes that allow iPOJO handlers to employ introspective techniques. iPOJO exploits meta-information of OSGi bundles in order to trigger actions according to method invocations and filed accesses. This is a powerful mechanism to implement dynamic adaptation techniques due to its capacity of performing actions based on information which is only available at runtime. Last, once an iPOJO handler is implemented, it is only necessary to define its XML scheme to make it available for iPOJO components.

In Figure 3, the `XtreemOSSelfAdaptableHandler` class represents how the XtreemOS iPOJO handler can be implemented. The Dynaco adaptation model is addressed by the `PrimitiveSelfAdaptableHandler` which separately deals with monitoring, decision, planning and execution concerns. This class should be used to implement generic adaptation mechanisms as decision-making engines. Thereby, each iPOJO self-adaptable handler ought to inherit this class by customizing it with domain-specific adaptation knowledge. With respect to the XtreemOS handler, this customization is represented by the `XtreemOSSelfAdaptableHandler` class. Finally, iPOJO flexible design allows iPOJO handlers to be used by each other, thus taking further advantages of re-

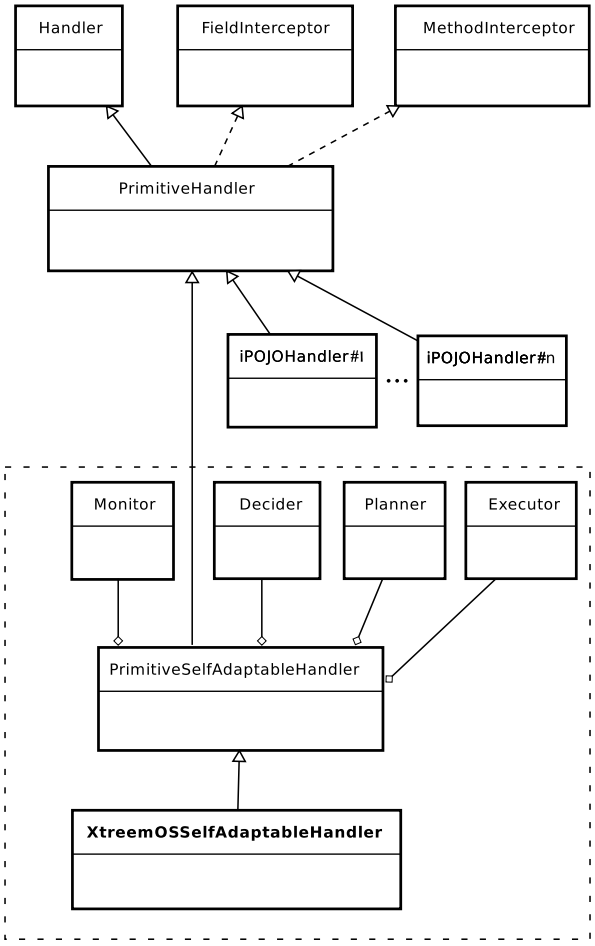

Fig. 3. Diagram class of the XtreemOS Self-Adaptable iPOJO Handler.

usability as we propose to use the iPOJO Event Admin handler when implementing the monitoring system.

C. Usage

The use of the XtreemOS iPOJO Handler is based on the *Job Submission Description Language* (JSDL) [2] and requires proper XtreemOS certificates. The JSDL relies on the XML standard to define the job and describe its resource requirements. Listing 1 exposes an example of a JSDL job. The *Executable* attribute at line 10 points to the program that comprises the job. It is followed by its arguments at line 11, the output and error files at lines 12 and 13. At line 18, we can realize that it was chosen 1 as the number of resources to run the job. Furthermore, with regard to security issues on XtreemOS, it uses X.509 certificates whose public keys must be informed to let the grid identify the user and grant right permission access on grid resources.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <JobDefinition xmlns:jsdl="http://schemas.ggf.org/jsdl
      /2005/11/jsdl">
3   <JobDescription>
```

```
4       <JobIdentification>
5           <Description>A simple grid job that lists
                current processes.</Description>
6           <JobProject>A Project Name.</JobProject>
7         </JobIdentification>
8         <Application>
9           <POSIXApplication xmlns:ns1="http://schemas.
                ggf.org/jsdl/2005/11/jsdl-posix">
10            <Executable>/bin/ps</Executable>
11            <Argument>-aef</Argument>
12            <Output>psx.out</Output>
13            <Error>psx.err</Error>
14          </POSIXApplication>
15        </Application>
16        <Resources>
17            <TotalResourceCount>
18                <Exact>1</Exact>
19            </TotalResourceCount>
20        </Resources>
21    </JobDescription>
22</JobDefinition>
```

Listing 1. A simple example of a grid job which is based on the JSDL standard.

As iPOJO component descriptions are addressed in the component `metada.xml` file, we propose to add there the job (i.e., a path to the JSDL file) as well as the XtreemOS user certificate and the job execution QoS. In Listing 2, there is an example of such a metadata file. In order to use the XtreemOS iPOJO Handler, the component requires it at line 2. At line 6, it is defined which class implements the component. Line 8 and 9 define the use of the XtreemOS iPOJO Handler and the job to be submitted respectively. As follows, some QoS are required such as a fault-tolerant execution that relies on three job execution replicas (line 11) whose data should also be twice replicated and stored in a persistent way (lines 12 and 13).

```
1<ipojo
2  xmlns:grid="org.apache.felix.ipojo.handlers.xtreemos">
3
4<component
5  name="MyComponentService"
6  classname="org.apache.felix.ipojo.sample.MyComponentImpl">
7
8  <grid:xtreemos
9    file="/home/alage/my_jobs/job.jsdl"
10   credentials="/home/alage/.xos/truststore/certs/user.crt"
11   FTExecution="3"
12   dataReplication="2"
13   dataPersistence="yes"/>
14
15</component>
16
17<instance
18  name="AnInstanceOfMyComponentService"
19  component="MyComponentService"/>
20
21</ipojo>
```

Listing 2. Example of an iPOJO component `metadata.xml` which configures the XtreemOS iPOJO Handler usage.

VII. Related Work

Some approaches have targeted the development of grid-oriented services [11], [9]. In [11], the authors propose an approach for automatically managing grid applications. They rely on SOA standards as Web Services to define the communication among grid functionalities. In [9], the authors propose an architecture that translates high-level service requirements to resource-level requirements. They provide an automatic way of negotiating both requirements by establishing contracts which define specific QoS. However, both approaches do not target to ease the development of grid-oriented services as we do. In other words, [11] focus on providing automatic mechanisms for managing grid applications and [9] aim at a autonomous architecture for translating and negotiating QoS. They have such specific goals rather than providing a solution that brings together both service development easing and job self-management.

Ultimately, the Web Service Resource Framework [5] addresses a grid interface driven to Web Services. It adds the idea of stateful Web Services that describes grid resource states in order to manage their life cycle. Despite the fact that it enables grids to conceive service-based applications, the job management is still the service developer charge. In contrast, we propose to automatically manage the job submission by using the component-based design and an adaptation model that eases the conception and maintenance of dynamic and flexible services.

VIII. Conclusion and Future Work

This work presents an approach that aims at easing the development of grid-oriented services. We propose a self-adaptable architecture that enables the automatic management of job submissions and considers QoS related to the job execution. Such an architecture leverages the iPOJO component model to enable the conception of OSGi services that use the grid. An iPOJO handler is proposed by leveraging Dynaco (Dynamic Adaptation for Components) to automatically address the self-adaptable job management on the XtreemOS grid infrastructure.

Furthermore, the scope of our approach is confined to services which must be aware of grids. By that very fact and intending to ease the development of such services, we put together dynamic and flexible service development and job self-management. On one hand, it presents an interesting approach for grid-oriented services. On the other hand, our proposal enforces service developers to deal with grid jobs by exposing the grid infrastructure to other service developers that do not need to execute grid jobs but do need a distributed, large-scale and heterogeneous infrastructure for executing services. In future works, we will investigate how generic services can be executed using grids. That leads us to still exploit job self-management but in such a way that services could be executed with no knowledge of grids.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-CUBE).

REFERENCES

[1] M. Aksit and Z. Choukair. Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 84–89, Washington, DC, USA, May 2003.

[2] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. GFD-R.056 - Job Submission Description Language (JSDL) Specification. Technical report, Global Grid Forum, 2005.

[3] J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for Grid computing. In *EGC '05: Proceedings of The European Grid Conference*, pages 538–547, Amsterdam, June 2005.

[4] H. Cervantes and R. S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.

[5] K. Czajkowski, D. Ferguson, I. Foster, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution. Technical report, Fujitsu Limited and International Business Machines Corporation and The University of Chicago, May 2004.

[6] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. *SCC '07: Proceedings of The IEEE International Conference on Services Computing*, pages 474–481, July 2007.

[7] I. Foster. What is the Grid? - A Three Point Checklist. *GRIDtoday*, 1:22–25, 2002.

[8] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith. A Simple API for Grid Applications (SAGA). Global Grid Forum, January 2008.

[9] P. Hasselmeyer, B. Koller, L. Schubert, and P. Wieder. Towards SLA-Supported Resource Management. In *HPCC '06: Proceedings of the 2006 International Conference on High Performance Computing and Communications*, pages 743–752. Springer, 2006.

[10] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.

[11] H. Liu, V. Bhat, M. Parashar, and S. Klasky. An autonomic service architecture for self-managing grid applications. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 132–139, Washington, DC, USA, 2005. IEEE Computer Society.

[12] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, 2004.

[13] Open Service Oriented Architecture. SCA Service Component Architecture. Assembly Model Specification, March 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

[14] OSGi Alliance. OSGi Service Platform Core Specification. Release 4, Version 4.1, April 2007.

[15] OSGi Alliance. OSGi Service Platform Service Compendium. Release 4, Version 4.1, April 2007. Pages 281–315.

[16] OSGi Alliance. OSGi Service Platform Release 4. Version 4.2 (Early Draft 3), March 2009.

[17] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing, Introduction. *Commun. ACM*, 46(10):24–28, 2003.

[18] C. Szyperski. Component Technology: What, Where, and How? In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society.

[19] W3C Working Group. Web Services Architecture. http://www.w3.org/TR/ws-arch/, 2010.

[20] J. Yang. Web Service Componentization. *Communications of the ACM*, 46(10):35–40, 2003.